An Introduction to VHDL Based Design for Xilinx FPGAs
1 Introduction

The purpose of this lab is to give a first introduction to the VHDL based design flow for Xilinx FPGAs. The entire VHDL source is given to you, but it is highly recommended that you study the code and make sure that you understand the main concepts.

Note that you can only learn a subset of the tools in this first tutorial-based lab. As you work through future labs, it is expected that you learn the tools by experimenting with the features/options. When you run into problems, or have any questions, please ask for help. However it is usually a good idea to study the extensive online documentation. You should under no circumstances try to contact technical support for any software or hardware used during the course.

To get this lab accepted you have to answer and turn in some questions. The questions are available in a separate document.

1.1 The design flow

For design entry Emacs will be used. Simulation will be performed with Cadence NC VHDL and the design will be synthesized with Synplify. The Xilinx Alliance software will be used for design implementation. The design flow is illustrated in figure 1.

Design entry

Traditionally, design entry has been performed using schematic entry tools. As designs become more complex Hardware Description Languages (HDL), such as VHDL and Verilog, have been adopted. HDLs allows designers to describe hardware using abstract programming.

For beginners it might be a bit hard to understand VHDL and how the hardware is created from the code. One recommendation is to “think hardware” when you write your VHDL code. It is not true that when you know VHDL you are also automatically a good digital designer. You should regard VHDL as a “tool” to simplify the design work - you have to understand VHDL and digital design.

VHDL is a complex and feature rich language. During this course you will, mostly on your own, only learn a subset of the language. VHDL was initially a language for describing the function and structure of digital systems, not for synthesis. That is the reason why only a subset of VHDL is synthesizable. So for logic synthesis you do not have to understand the complete VHDL.

In the labs you will use Emacs to write your VHDL code. However if you prefer any other text editor, you may use that.
Figure 1. Design Flow

**Functional simulation**

Functional simulation verifies that your VHDL model behaves as expected. At this stage you have no timing information about your design. Do not try to model delays in VHDL that is written for synthesis. The timing information will come from your technology library once the design is implemented.
Synthesis

Synthesis may be defined as constructing a target technology specific netlist from a model of a circuit described in VHDL, or any other HDL.

In the Synplify manual we find this description of synthesis:

“Although commonly called logic synthesis, or just synthesis, there are several steps going on behind the scenes in a synthesis tool program. The first step is language synthesis (HDL compilation) in which the design you described at a high level of abstraction is compiled to known structural elements. The second step is optimization, in which algorithms make your design as small as possible and run faster. Synplify can perform optimizations with no knowledge of the target technology, while the remainder of the optimizations are performed during the final step. The final step is technology mapping in which the synthesis tool maps your design, using architecture-specific techniques. A design netlist is then generated in a format readable by your programmable logic vendor’s place and route tool.”

Design implementation

The netlist generated by the synthesis tool is the input to the Xilinx Alliance tools.

- **Translation** Converts the netlist into a tool specific binary format.
- **Mapping** Assigns a design’s logic elements to the specific physical elements, such as CLBs and IOBs, that actually implement logic functions in a device.
- **Placing** Assigns design blocks created during mapping to specific locations in the FPGA.
- **Routing** Assigns the interconnect paths in the FPGA.
- **Static timing analysis** Analysis of delays in the implemented design. Determines if the timing constraints have been met.
- **Bitstream generation** Converts a design to a bitstream that may be used to configure the FPGA.
- **Back annotation** The implemented design is converted to a VHDL netlist and the net delays are extracted to a Standard Delay Format (SDF) file.

Timing simulation

The timing simulation is performed to verify that the function is correct and that there are no timing violations in the implemented design. A structural VHDL netlist and a SDF file is loaded in NC VHDL and the same simulation may be performed as was performed prior to synthesis.

For a fully synchronous design the static timing analysis should cover all timing constraints, but a static timing analysis does not verify the functionality. In this lab we will not perform a timing simulation, but in later labs we will.
2 The design

The design, illustrated in figure 2, in this first lab is very simple. A larger view of the figure can be found at the end of this document.

![Figure 2. Block diagram of the design.](image)

The design is a counter that counts from 00 – 99. The current count is shown on two LED digits and the state of the least significant digit is also displayed on a ten-segment bargraph led. For instance when the least significant digit is ‘4’ then the forth segment on the bargraph LED is lit.

There are four pushbuttons, SW1-SW4, that control the counter. SW1 is used to synchronously reset the counter. SW2 is used to start or stop the counter depending on its current state. SW3 and SW4 are used to control the speed of the counter, increase and decrease the speed. The pushbuttons need to be debounced inside the FPGA, however we don’t care to debounce the reset input.

The “Start_Stop” signal controls a register, “Go-state” in figure 2, that holds the current state of the counter (running or stopped). The output from the register is called “Go” and is active high, i.e. “Go” is high when the counter is running.

Not shown in figure 2 is the global clock that is set to 20 MHz. Since this rate is a “bit” too high for us to able to see LEDs actually count, a Numerically Controlled Oscillator (NCO) is used to create a signal that enables the counter flips-flops at a lower rate. It would have been possible to use a lower clock rate, but for practical reasons we don’t want to change the clock rate during the course.

The NCO enables a counter, Count9, and a one-hot counter. The counter holds the state of the least significant digit and the one-hot counter drives the bargraph led. When the counter for the least significant digit wraps around (goes from 9 to 0) a pulse is generated that enables the most significant digit counter.

The “Bin2led” boxes in figure 2 converts the unsigned format outputs from the counters so the corresponding digit will be displayed on the LEDs.

Following a few of the modules will be described in more detail.

---

1 The name NCO is perhaps a bit misleading. A NCO is often used to generate waveforms such as sine and cosine. In this case we use an NCO to generate pulses.
2.1 NCO

As previously mentioned the NCO is used to reduce the rate of the counters. A block diagram of the NCO is shown in figure 3.

The input signals “Faster” and “Slower” controls an 8 bit counter. If “Faster” is high and “Slower” is low the counter is incremented and if “Slower” is high and “Faster” is low the counter is decremented, otherwise the counter retains its state. The output, “Ref” from the counter feeds another 26 bit counter.

When the wider counter generates a carry a pulse is generated. The length of the pulse is limited to one clock cycle by the aid of a single register and an AND gate with one inverting input. The rate of the “Pulse” output is defined by

\[ f = \frac{\text{Ref} \cdot f_{\text{clock}}}{2^N} \]

where Ref is an integer in the range \([0, 2^8 - 1]\), \(f_{\text{clock}} = 20 \text{ MHz}\) and \(N = 26\).

2.2 De-bouncer

The pushbuttons on the XSV board are not de-bounced, so the de-bouncing has to be done internally in the FPGA. When a pushbutton is pressed it does not output a clean pulse instead the output may toggle as illustrated in figure 4.

The function of the de-bouncer is that it should output a pulse that is one clock cycle wide each time a button is pushed. In order to output another pulse the button must be released and pressed again.

The de-bouncer module consists of a 17-bit counter and a two state Finite State Machine (FSM). When the FPGA is initially powered up or reset the state of the FSM is Active, which means that the de-bouncer is waiting for the pushbutton to be pressed. When the button is pressed the 17-bit counter starts to count. The counter is reset whenever the pushbutton is released. This means
that counter will generate a carry only if it is enabled for $2^{17}$ clock cycles, which is about 7 ms. The carry is the output of the de-bouncer. In order to avoid that the output pulses more than once each time the pushbutton is pressed the FSM enters the Idle state where the counter is always reset. The pushbutton must be released in order for the FSM to enter the Active state again.

3 Functional simulation

You will now use NC VHDL to verify that the design is functionally correct. Actually we will not simulate the de-bouncers. We will bypass the de-bouncers by setting a Boolean generic “Debug”, in the testbench for the “Lab1” module. By setting the generic the width of the NCO is also reduced to 10 bits. The reason why we do this is that is simply that this speeds up the simulation significantly. For instance the NCO divides the clock by a factor up to $2^{26}$ times. So you would have to simulate $2^{26}$ clock cycles before you even see a change on the counter output. But, don’t worry, I have simulated the NCO and de-bouncer separately and made sure that they are ok.

3.1 Preparing your simulation environment

Refer to the NC VHDL tutorial on how to create the cds.lib and hdl.var files

3.2 Compiling the code

In the NC VHDL tutorial you learned how to compile the VHDL code using the command line and the Launchtool. You may also compile the code in Emacs. Emacs has a powerful VHDL mode, which I highly recommend that you use. However here we will use the command line to compile the source code.

Figure 5 illustrates the project hierarchy and indicates in which order the different files should be compiled. Packages must be compiled before they are referenced in any other design unit, so smd098_pkg.vhd must be compiled first. After that the different modules are compiled from bottom to up in the hierarchy.

![Figure 5. Project hierarchy](image)

Compile the VHDL files with NC VHDL in the correct order. Compile the files as you learned in the tutorial but add the switch `-v93`, which enables VHDL ’93 features.
3.3 Elaborating the design

Elaborate the top level for simulation which is `worklib.lab1_tb:sim`. Use the `-v93` for elaboration as well.

3.4 Simulating the design

Invoke the simulator with the command:

```bash
csim -gui worklib.lab1_tb:sim
```

In the test bench add the signals “LeftLed”, “RightLed” and “Bar” to the wave window. The two first signals are the LED digit outputs decoded into integers. In the I/O region of the simulator type the command

```bash
run 11 ms
```

Now examine the waveform and make sure that the counter is behaving as expected – it should count from 00 to 99.

Note that this test bench is very simple and the simulation you just performed can not be considered to be a full test. You are encouraged to explore the simulator in more detail. You will definitely use the simulator more thoroughly in later labs.

4 Synthesis

Start Synplify PRO and create a new project called Lab1. Add the VHDL files as indicated in the figure below and set the implementation options as indicated:

You should of course not add the testbench!

Please note that figures are from an earlier version of Synplify and that the GUI is updated, but this should not cause any problems. Set the same options!
Next press F7 to “Compile Only”. Then create a constraint file using the same procedure as in the Synplify tutorial. Constrain the clock to 20 MHz and set the input/output delays as illustrated below:

Read this carefully: The constraints means that the input signals are valid 15 ns after previous rising edge of the clock and that the output signals must be available 10 ns before the next rising edge of the clock. Save the constraints and add the constraint file to the project!

Now complete the synthesis!

5 Design implementation

The output from synthesis is an EDIF netlist that represents the design primitives and its interconnection. The Xilinx Alliance tools will process the netlist in order to create a configuration file for the FPGA. Apart from the netlist, you have been provided with a user constraint file (UCF). The constraints in the UCF assigns the input and output pads to specific package pins.

To start the Xilinx Design Manager type:

xilinx

Select File → New Project. In the dialog box that appears the “Input Design” should be the EDIF file lab1.edf located in your rev1 directory and the “Work Directory” should be rev1/xproj.
After you click **OK** another dialog box will appear:

Set **Constraint File:** **Custom** and locate `virtex.ucf` – it should be in the same directory as you have your VHDL files. Let the other settings be the default. Click **OK**.

In the Design Manager window select **Design → Options**, which will display another dialog:

Change the **Place and Route Effort Level** to highest effort. In this lab we will **not** perform a back-end timing simulation so the **Simulation** option should be **OFF**. Click **Implementation:** **Edit Options** and change the **Timing Report** options as indicated below.
In the Design Manager window select **Design → Implement** to start the implementation process. The Flow Engine window will appear.

When the implementation process has finished you will return to the Design Manager window and select **Utilities → Report Browser** to display:
Open and read all reports!

There are some other tools that you will investigate on your own. From the Tools menu you should start FPGA Editor and Floorplanner.

6 Testing the design on the demoboard

There are two XSV100 boards available in the hardware lab and they should both be setup and ready for use. Make sure that the ATX power supply is on and check the parallel port connection.

Please be careful when you handle the boards. The boards are placed on an ESD protected surface but you don't have to wear any arm wrist. Just avoid touching the components on the board.

Start the tool GXSLoad with the shortcuts in the Windows start menu in XSTools.

Locate the lab1.bit file in your /rev1/xproj/ver1/rev1 directory with explorer. Drag and drop the file in the GXSLoad window. When the download is completed press the pushbuttons to verify that the design is working.

Now you are finished with the practical part of this lab, but don’t forget to complete the questions and mail them to Jonas.